上文介绍了Android应用程序的启动过程，即应用程序默认Activity的启动过程，一般来说，这种默认Activity是在新的进程和任务中启动的；本文将继续分析在应用程序内部启动非默认Activity的过程的源代码，这种非默认Activity一般是在原来的进程和任务中启动的。

        这里，我们像上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)一样，采用再上一篇文章[Android应用程序的Activity启动过程简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/6685853)所举的例子来分析在应用程序内部启动非默认Activity的过程。

        在应用程序内部启动非默认Activity的过程与在应用程序启动器Launcher中启动另外一个应用程序的默认Activity的过程大体上一致的，因此，这里不会像上文[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)一样详细分析每一个步骤，我们着重关注有差别的地方。

        回忆一下[Android应用程序的Activity启动过程简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/6685853)一文所用的应用程序Activity，它包含两个Activity，分别是MainActivity和SubActivity，前者是应用程序的默认Activity，后者是非默认Activity。MainActivity启动起来，通过点击它界面上的按钮，便可以在应用程序内部启动SubActivity。

        我们先来看一下应用程序的配置文件AndroidManifest.xml，看看这两个Activity是如何配置的：

**[html]** [view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/6703247)

1. **<?xml** version="1.0" encoding="utf-8"**?>**
2. **<manifest** xmlns:android="http://schemas.android.com/apk/res/android"
3. package="shy.luo.activity"
4. android:versionCode="1"
5. android:versionName="1.0"**>**
6. **<application** android:icon="@drawable/icon" android:label="@string/app\_name"**>**
7. **<activity** android:name=".MainActivity"
8. android:label="@string/app\_name"**>**
9. **<intent-filter>**
10. **<action** android:name="android.intent.action.MAIN" **/>**
11. **<category** android:name="android.intent.category.LAUNCHER" **/>**
12. **</intent-filter>**
13. **</activity>**
14. **<activity** android:name=".SubActivity"
15. android:label="@string/sub\_activity"**>**
16. **<intent-filter>**
17. **<action** android:name="shy.luo.activity.subactivity"**/>**
18. **<category** android:name="android.intent.category.DEFAULT"**/>**
19. **</intent-filter>**
20. **</activity>**
21. **</application>**
22. **</manifest>**

<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="shy.luo.activity"

android:versionCode="1"

android:versionName="1.0">

<application android:icon="@drawable/icon" android:label="@string/app\_name">

<activity android:name=".MainActivity"

android:label="@string/app\_name">

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

<activity android:name=".SubActivity"

android:label="@string/sub\_activity">

<intent-filter>

<action android:name="shy.luo.activity.subactivity"/>

<category android:name="android.intent.category.DEFAULT"/>

</intent-filter>

</activity>

</application>

</manifest>

        这里可以很清楚地看到，MainActivity被配置成了应用程序的默认Activity，而SubActivity可以通过名称“shy.luo.activity.subactivity”隐式地启动，我们来看一下src/shy/luo/activity/MainActivity.java文件的内容，可以清楚地看到SubActivity是如何隐式地启动的：

**[java]** [view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/6703247)

1. public class MainActivity extends Activity  implements OnClickListener {
2. ......
4. @Override
5. public void onClick(View v) {
6. if(v.equals(startButton)) {
7. Intent intent = new Intent("shy.luo.activity.subactivity");
8. startActivity(intent);
9. }
10. }
11. }

public class MainActivity extends Activity implements OnClickListener {

......

@Override

public void onClick(View v) {

if(v.equals(startButton)) {

Intent intent = new Intent("shy.luo.activity.subactivity");

startActivity(intent);

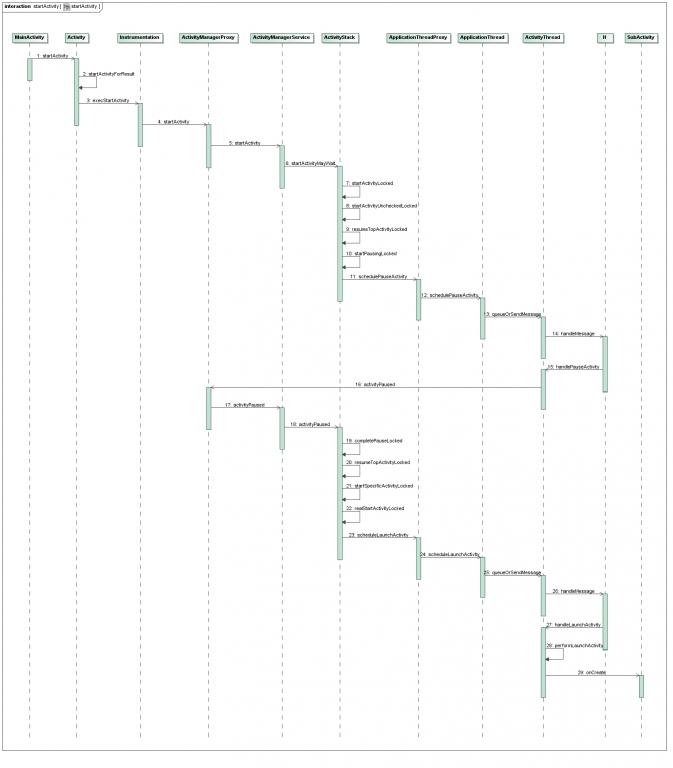
}

}

}

       这里，首先创建一个名称为“shy.luo.activity.subactivity”的Intent，然后以这个Intent为参数，通过调用startActivity函数来实现隐式地启动SubActivity。

       有了这些背景知识后，我们就来看一下SubActivity启动过程的序列图：



[点击查看大图](http://hi.csdn.net/attachment/201108/19/0_13137647026lR7.gif)

       与前面介绍的MainActivity启动过程相比，这里少了中间创建新的进程的步骤；接下来，我们就详细分析一下SubActivity与MainActivity启动过程中有差别的地方，相同的地方请参考[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)一文。

       Step 1. Activity.startActivity

       这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 2大体一致，通过指定名称“shy.luo.activity.subactivity”来告诉应用程序框架层，它要隐式地启动SubActivity。所不同的是传入的参数intent没有Intent.FLAG\_ACTIVITY\_NEW\_TASK标志，表示这个SubActivity和启动它的MainActivity运行在同一个Task中。

       Step 2. Activity.startActivityForResult

       这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 3一致。

       Step 3. Instrumentation.execStartActivity

       这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 4一致。

       Step 4. ActivityManagerProxy.startActivity

       这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 5一致。

       Step 5. ActivityManagerService.startActivity

       这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 6一致。

       Step 6. ActivityStack.startActivityMayWait

       这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 7一致。

       Step 7. ActivityStack.startActivityLocked

       这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 8一致。

       Step 8. ActivityStack.startActivityUncheckedLocked

       这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 9有所不同，主要是当前要启动的Activity与启动它的Activity是在同一个Task中运行的，我们来详细看一下。这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

**[java]** [view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/6703247)

1. public class ActivityStack {
3. ......
5. final int startActivityUncheckedLocked(ActivityRecord r,
6. ActivityRecord sourceRecord, Uri[] grantedUriPermissions,
7. int grantedMode, boolean onlyIfNeeded, boolean doResume) {
8. final Intent intent = r.intent;
9. final int callingUid = r.launchedFromUid;
11. int launchFlags = intent.getFlags();
13. ......
15. if (sourceRecord == null) {
16. ......
17. } else if (sourceRecord.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {
18. ......
19. } else if (r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE
20. ......
21. }
23. if (r.resultTo != null && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {
24. ......
25. }
27. boolean addingToTask = false;
28. if (((launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0 &&
29. (launchFlags&Intent.FLAG\_ACTIVITY\_MULTIPLE\_TASK) == 0)
30. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK
31. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {
32. ......
33. }
35. if (r.packageName != null) {
36. // If the activity being launched is the same as the one currently
37. // at the top, then we need to check if it should only be launched
38. // once.
39. ActivityRecord top = topRunningNonDelayedActivityLocked(notTop);
40. if (top != null && r.resultTo == null) {
41. if (top.realActivity.equals(r.realActivity)) {
42. ......
43. }
44. }
46. } else {
47. ......
48. }
50. boolean newTask = false;
52. // Should this be considered a new task?
53. if (r.resultTo == null && !addingToTask
54. && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {
55. ......
57. } else if (sourceRecord != null) {
58. ......
59. // An existing activity is starting this new activity, so we want
60. // to keep the new one in the same task as the one that is starting
61. // it.
62. r.task = sourceRecord.task;
63. ......
65. } else {
66. ......
67. }
69. ......
71. startActivityLocked(r, newTask, doResume);
72. return START\_SUCCESS;
73. }
75. ......
77. }

public class ActivityStack {

......

final int startActivityUncheckedLocked(ActivityRecord r,

ActivityRecord sourceRecord, Uri[] grantedUriPermissions,

int grantedMode, boolean onlyIfNeeded, boolean doResume) {

final Intent intent = r.intent;

final int callingUid = r.launchedFromUid;

int launchFlags = intent.getFlags();

......

if (sourceRecord == null) {

......

} else if (sourceRecord.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {

......

} else if (r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE

......

}

if (r.resultTo != null && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {

......

}

boolean addingToTask = false;

if (((launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0 &&

(launchFlags&Intent.FLAG\_ACTIVITY\_MULTIPLE\_TASK) == 0)

|| r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK

|| r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {

......

}

if (r.packageName != null) {

// If the activity being launched is the same as the one currently

// at the top, then we need to check if it should only be launched

// once.

ActivityRecord top = topRunningNonDelayedActivityLocked(notTop);

if (top != null && r.resultTo == null) {

if (top.realActivity.equals(r.realActivity)) {

......

}

}

} else {

......

}

boolean newTask = false;

// Should this be considered a new task?

if (r.resultTo == null && !addingToTask

&& (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {

......

} else if (sourceRecord != null) {

......

// An existing activity is starting this new activity, so we want

// to keep the new one in the same task as the one that is starting

// it.

r.task = sourceRecord.task;

......

} else {

......

}

......

startActivityLocked(r, newTask, doResume);

return START\_SUCCESS;

}

......

}

        这里，参数intent的标志位Intent.FLAG\_ACTIVITY\_NEW\_TASK没有设置，在配置文件AndriodManifest.xml中，SubActivity也没有配置启动模式launchMode，于是它就默认标准模式，即ActivityInfo.LAUNCH\_MULTIPLE，因此，下面if语句不会执行：

**[java]** [view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/6703247)

1. if (((launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0 &&
2. (launchFlags&Intent.FLAG\_ACTIVITY\_MULTIPLE\_TASK) == 0)
3. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK
4. || r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {
5. ......
6. }

if (((launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0 &&

(launchFlags&Intent.FLAG\_ACTIVITY\_MULTIPLE\_TASK) == 0)

|| r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_TASK

|| r.launchMode == ActivityInfo.LAUNCH\_SINGLE\_INSTANCE) {

......

}

        于是，变量addingToTask为false。

         继续往下看：

**[java]** [view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/6703247)

1. if (r.packageName != null) {
2. // If the activity being launched is the same as the one currently
3. // at the top, then we need to check if it should only be launched
4. // once.
5. ActivityRecord top = topRunningNonDelayedActivityLocked(notTop);
6. if (top != null && r.resultTo == null) {
7. if (top.realActivity.equals(r.realActivity)) {
8. ......
9. }
10. }
12. }

if (r.packageName != null) {

// If the activity being launched is the same as the one currently

// at the top, then we need to check if it should only be launched

// once.

ActivityRecord top = topRunningNonDelayedActivityLocked(notTop);

if (top != null && r.resultTo == null) {

if (top.realActivity.equals(r.realActivity)) {

......

}

}

}

        这里看一下当前要启动的Activity是否就是当前堆栈顶端的Activity，如果是的话，在某些情况下，就不用再重新启动了。函数topRunningNonDelayedActivityLocked返回当前  
堆栈顶端的Activity，这里即为MainActivity，而当前要启动的Activity为SubActivity，因此，这二者不相等，于是跳过里面的if语句。

        接着往下执行：

**[java]** [view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/6703247)

1. // Should this be considered a new task?
2. if (r.resultTo == null && !addingToTask
3. && (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {
4. ......
6. } else if (sourceRecord != null) {
7. ......
8. // An existing activity is starting this new activity, so we want
9. // to keep the new one in the same task as the one that is starting
10. // it.
11. r.task = sourceRecord.task;
12. ......
14. } else {
15. ......
16. }

// Should this be considered a new task?

if (r.resultTo == null && !addingToTask

&& (launchFlags&Intent.FLAG\_ACTIVITY\_NEW\_TASK) != 0) {

......

} else if (sourceRecord != null) {

......

// An existing activity is starting this new activity, so we want

// to keep the new one in the same task as the one that is starting

// it.

r.task = sourceRecord.task;

......

} else {

......

}

        前面说过参数intent的标志位Intent.FLAG\_ACTIVITY\_NEW\_TASK没有设置，而这里的sourceRecord即为当前执行启动Activity操作的Activity，这里即为MainActivity，因此，它不为null，于是于MainActivity所属的Task设置到r.task中去，这里的r即为SubActivity。看到这里，我们就知道SubActivity要和MainActivity运行在同一个Task中了，同时，变量newTask的值为false。

        最后，函数进 入startActivityLocked(r, newTask, doResume)进一步处理了。这个函数同样是定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

**[java]** [view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/6703247)

1. public class ActivityStack {
3. ......
5. private final void startActivityLocked(ActivityRecord r, boolean newTask,
6. boolean doResume) {
7. final int NH = mHistory.size();
9. int addPos = -1;
11. if (!newTask) {
12. // If starting in an existing task, find where that is...
13. boolean startIt = true;
14. for (int i = NH-1; i >= 0; i--) {
15. ActivityRecord p = (ActivityRecord)mHistory.get(i);
16. if (p.finishing) {
17. continue;
18. }
19. if (p.task == r.task) {
20. // Here it is!  Now, if this is not yet visible to the
21. // user, then just add it without starting; it will
22. // get started when the user navigates back to it.
23. addPos = i+1;
24. if (!startIt) {
25. mHistory.add(addPos, r);
26. r.inHistory = true;
27. r.task.numActivities++;
28. mService.mWindowManager.addAppToken(addPos, r, r.task.taskId,
29. r.info.screenOrientation, r.fullscreen);
30. if (VALIDATE\_TOKENS) {
31. mService.mWindowManager.validateAppTokens(mHistory);
32. }
33. return;
34. }
35. break;
36. }
37. if (p.fullscreen) {
38. startIt = false;
39. }
40. }
41. }
43. ......
45. // Slot the activity into the history stack and proceed
46. mHistory.add(addPos, r);
47. r.inHistory = true;
48. r.frontOfTask = newTask;
49. r.task.numActivities++;
51. ......
53. if (doResume) {
54. resumeTopActivityLocked(null);
55. }
56. }
58. ......
60. }

public class ActivityStack {

......

private final void startActivityLocked(ActivityRecord r, boolean newTask,

boolean doResume) {

final int NH = mHistory.size();

int addPos = -1;

if (!newTask) {

// If starting in an existing task, find where that is...

boolean startIt = true;

for (int i = NH-1; i >= 0; i--) {

ActivityRecord p = (ActivityRecord)mHistory.get(i);

if (p.finishing) {

continue;

}

if (p.task == r.task) {

// Here it is! Now, if this is not yet visible to the

// user, then just add it without starting; it will

// get started when the user navigates back to it.

addPos = i+1;

if (!startIt) {

mHistory.add(addPos, r);

r.inHistory = true;

r.task.numActivities++;

mService.mWindowManager.addAppToken(addPos, r, r.task.taskId,

r.info.screenOrientation, r.fullscreen);

if (VALIDATE\_TOKENS) {

mService.mWindowManager.validateAppTokens(mHistory);

}

return;

}

break;

}

if (p.fullscreen) {

startIt = false;

}

}

}

......

// Slot the activity into the history stack and proceed

mHistory.add(addPos, r);

r.inHistory = true;

r.frontOfTask = newTask;

r.task.numActivities++;

......

if (doResume) {

resumeTopActivityLocked(null);

}

}

......

}

        这里传进来的参数newTask为false，doResume为true。当newTask为false，表示即将要启动的Activity是在原有的Task运行时，如果这个原有的Task当前对用户不可见时，这时候就不需要继续执行下去了，因为即使把这个Activity启动起来，用户也看不到，还不如先把它保存起来，等到下次这个Task对用户可见的时候，再启动不迟。这里，这个原有的Task，即运行MainActivity的Task当前对用户是可见的，因此，会继续往下执行。

        接下去执行就会把这个SubActivity通过mHistroy.add(addPos, r)添加到堆栈顶端去，然后调用resumeTopActivityLocked进一步操作。

        Step 9. ActivityStack.resumeTopActivityLocked

        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 10一致。

        但是要注意的是，执行到这个函数的时候，当前处于堆栈顶端的Activity为SubActivity，ActivityStack的成员变量mResumedActivity指向MainActivity。  
        Step 10. ActivityStack.startPausingLocked  
        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 11一致。

        从这里开始，ActivityManagerService通知MainActivity进入Paused状态。

        Step 11. ApplicationThreadProxy.schedulePauseActivity

        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 12一致。

        Step 12. ApplicationThread.schedulePauseActivity

        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 13一致。

        Step 13. ActivityThread.queueOrSendMessage  
        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 14一致。

        Step 14. H.handleMessage

        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 15一致。

        Step 15. ActivityThread.handlePauseActivity

        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 16一致。

        Step 16. ActivityManagerProxy.activityPaused

        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 17一致。

        Step 17. ActivityManagerService.activityPaused

        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 18一致。

        Step 18. ActivityStack.activityPaused

        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 19一致。

        Step 19. ActivityStack.completePauseLocked

        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 20一致。

        执行到这里的时候，MainActivity就进入Paused状态了，下面就开始要启动SubActivity了。

        Step 20. ActivityStack.resumeTopActivityLokced

        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 21一致。

        Step 21. ActivityStack.startSpecificActivityLocked

        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 22就有所不同了，这里，它不会调用mService.startProcessLocked来创建一个新的进程来启动新的Activity，我们来看一下这个函数的实现，这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

**[java]** [view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/6703247)

1. public class ActivityStack {
3. ......
5. private final void startSpecificActivityLocked(ActivityRecord r,
6. boolean andResume, boolean checkConfig) {
7. // Is this activity's application already running?
8. ProcessRecord app = mService.getProcessRecordLocked(r.processName,
9. r.info.applicationInfo.uid);
11. ......
13. if (app != null && app.thread != null) {
14. try {
15. realStartActivityLocked(r, app, andResume, checkConfig);
16. return;
17. } catch (RemoteException e) {
18. ......
19. }
20. }
22. ......
24. }
26. ......
28. }

public class ActivityStack {

......

private final void startSpecificActivityLocked(ActivityRecord r,

boolean andResume, boolean checkConfig) {

// Is this activity's application already running?

ProcessRecord app = mService.getProcessRecordLocked(r.processName,

r.info.applicationInfo.uid);

......

if (app != null && app.thread != null) {

try {

realStartActivityLocked(r, app, andResume, checkConfig);

return;

} catch (RemoteException e) {

......

}

}

......

}

......

}

        这里由于不是第一次启动应用程序的Activity（MainActivity是这个应用程序第一个启动的Activity），所以下面语句：

**[java]** [view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/6703247)

1. ProcessRecord app = mService.getProcessRecordLocked(r.processName,
2. nfo.applicationInfo.uid);

ProcessRecord app = mService.getProcessRecordLocked(r.processName,

r.info.applicationInfo.uid);

        取回来的app不为null。在上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)中，我们介绍过，在Activity应用程序中的AndroidManifest.xml配置文件中，我们没有指定application标签的process属性，于是系统就会默认使用package的名称，这里就是"shy.luo.activity"了。每一个应用程序都有自己的uid，因此，这里uid + process的组合就可以创建一个全局唯一的ProcessRecord。这个ProcessRecord是在前面启动MainActivity时创建的，因此，这里将它取回来，并保存在变量app中。注意，我们也可以在AndroidManifest.xml配置文件中指定SubActivity的process属性值，这样SubActivity就可以在另外一个进程中启动，不过很少有应用程序会这样做，我们不考虑这种情况。

        这个app的thread也是在前面启动MainActivity时创建好的，于是，这里就直接调用realStartActivityLocked函数来启动新的Activity了，新的Activity的相关信息都保存在参数r中了。

        Step 22. ActivityStack.realStartActivityLocked

        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 28一致。  
        Step 23. ApplicationThreadProxy.scheduleLaunchActivity

        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 29一致。

        Step 24. ApplicationThread.scheduleLaunchActivity

        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 30一致。

        Step 25. ActivityThread.queueOrSendMessage

        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 31一致。

        Step 26. H.handleMessage

        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 32一致。

        Step 27. ActivityThread.handleLaunchActivity

        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 33一致。

        Step 28. ActivityThread.performLaunchActivity

        这一步与上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)的Step 34一致，不过，这里要从ClassLoader里面加载的类就是shy.luo.activity.SubActivity了。

        Step 29. SubAcitiviy.onCreate

        这个函数定义在packages/experimental/Activity/src/shy/luo/activity/SubActivity.java文件中，这是我们自定义的app工程文件：

**[java]** [view plaincopyprint?](http://blog.csdn.net/luoshengyang/article/details/6703247)

1. public class SubActivity extends Activity implements OnClickListener {
3. ......
5. @Override
6. public void onCreate(Bundle savedInstanceState) {
7. ......
9. Log.i(LOG\_TAG, "Sub Activity Created.");
10. }
12. ......
14. }

public class SubActivity extends Activity implements OnClickListener {

......

@Override

public void onCreate(Bundle savedInstanceState) {

......

Log.i(LOG\_TAG, "Sub Activity Created.");

}

......

}

       这样，SubActivity就在应用程序Activity内部启动起来了。  
       在应用程序内部启动新的Activity的过程要执行很多步骤，但是整体来看，主要分为以下四个阶段：

       一. Step 1 - Step 10：应用程序的MainActivity通过Binder进程间通信机制通知ActivityManagerService，它要启动一个新的Activity；  
       二. Step 11 - Step 15：ActivityManagerService通过Binder进程间通信机制通知MainActivity进入Paused状态；  
       三. Step 16 - Step 22：MainActivity通过Binder进程间通信机制通知ActivityManagerService，它已经准备就绪进入Paused状态，于是ActivityManagerService就准备要在MainActivity所在的进程和任务中启动新的Activity了；  
       四. Step 23 - Step 29：ActivityManagerService通过Binder进程间通信机制通知MainActivity所在的ActivityThread，现在一切准备就绪，它可以真正执行Activity的启动操作了。

       和上一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)中启动应用程序的默认Activity相比，这里在应用程序内部启动新的Activity的过程少了中间创建新的进程这一步，这是因为新的Activity是在已有的进程和任务中执行的，无须创建新的进程和任务。

      这里同样不少地方涉及到了Binder进程间通信机制，相关资料请参考[Android进程间通信（IPC）机制Binder简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/6618363)一文。

      这里希望读者能够把本文和上文[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)仔细比较一下应用程序的默认Activity和非默认Activity启动过程的不同之处，以加深对Activity的理解。

      最后，在本文和上文中，我们多次提到了Android应用程序中任务（Task）的概念，它既不是我们在Linux系统中所理解的进程（Process），也不是线程（Thread），它是用户为了完成某个目标而需要执行的一系列操作的过程的一种抽象。这是一个非常重要的概念，它从用户体验的角度出发，界定了应用程序的边界，极大地方便了开发者利用现成的组件（Activity）来搭建自己的应用程序，就像搭积木一样，而且，它还为应用程序屏蔽了底层的进程，即一个任务中的Activity可以都是运行在同一个进程中，也中可以运行在不同的进程中。Android应用程序中的任务的概念，具体可以参考官方文档<http://developer.android.com/guide/topics/fundamentals/tasks-and-back-stack.html>，上面已经介绍的非常清楚了。